SystemC Configuration Tutorial

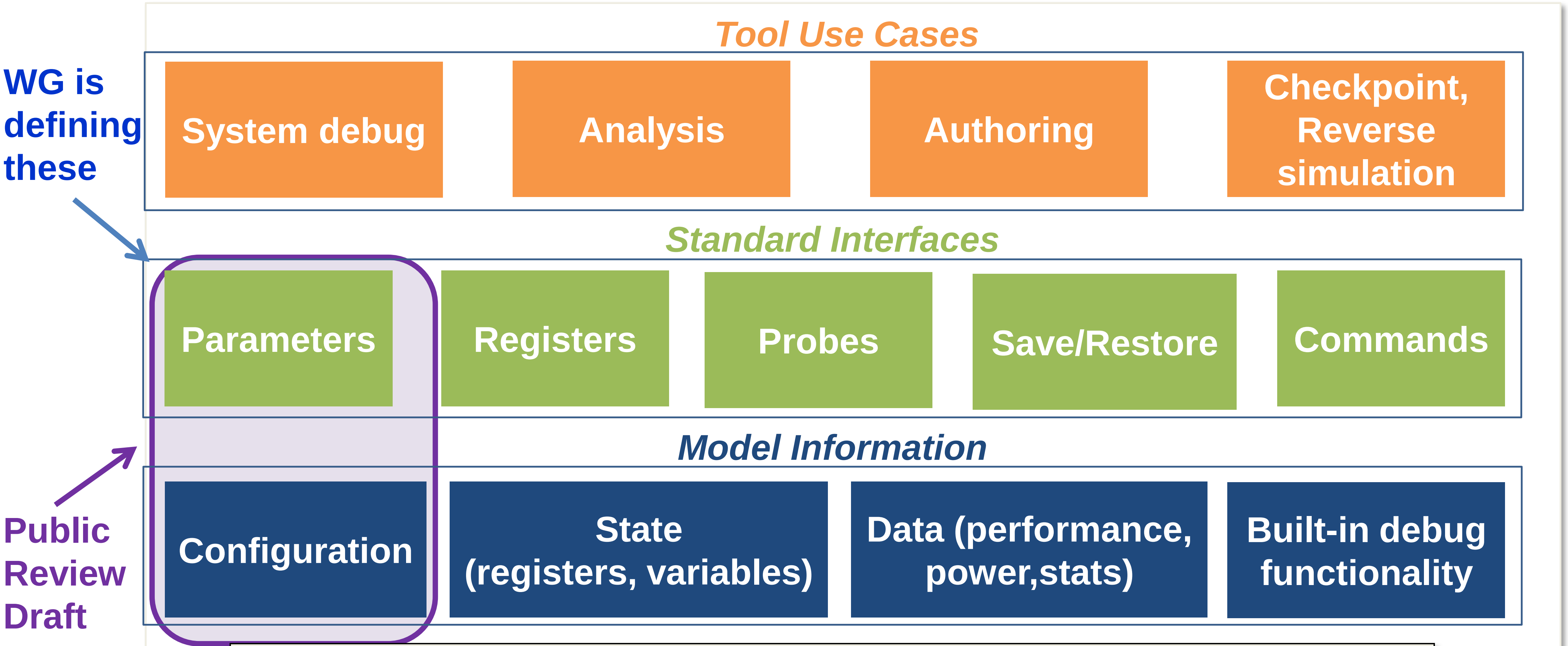
Public Review version of the draft standard

October 4, 2017

Acknowledgements

This tutorial content was contributed to the Accellera SystemC CCI WG by Ericsson, GreenSocs and Doulos.

Config, Control, Inspection WG



WG is defining these

Public Review Draft

Tool Use Cases

System debug

Analysis

Authoring

Checkpoint, Reverse simulation

Standard Interfaces

Parameters

Registers

Probes

Save/Restore

Commands

Model Information

Configuration

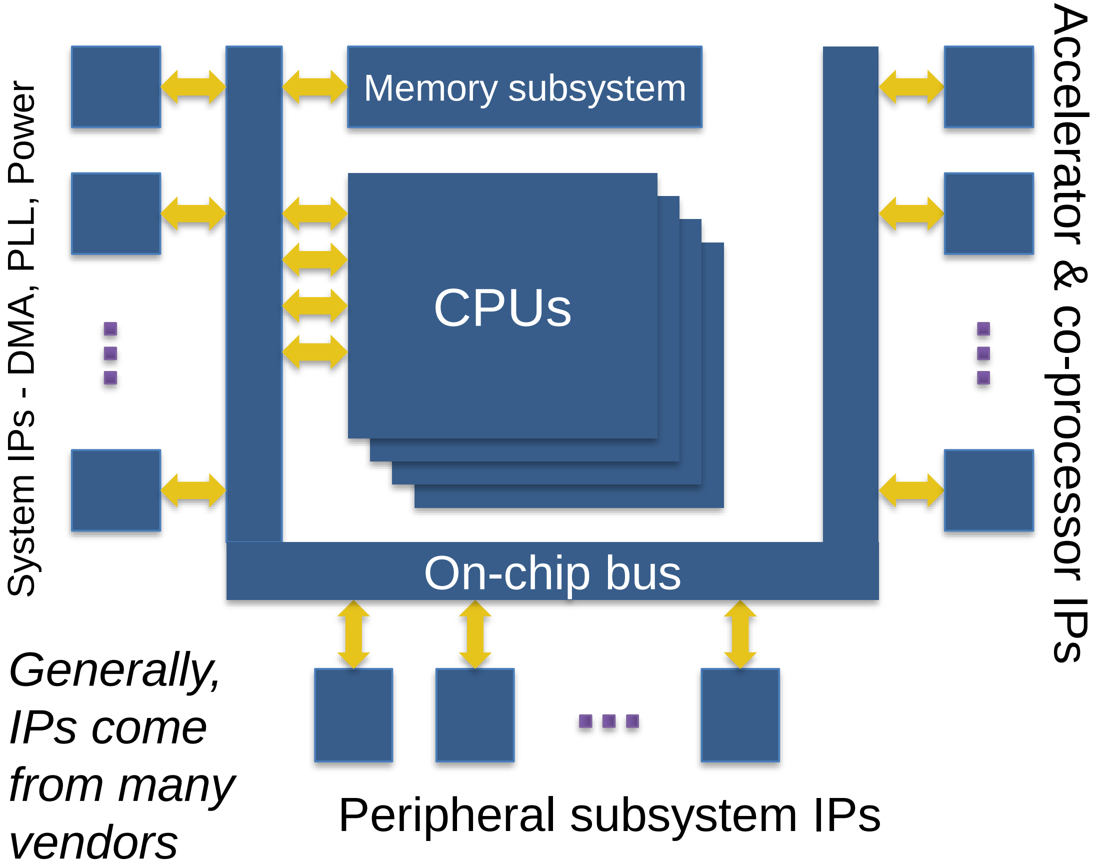
State (registers, variables)

Data (performance, power,stats)

Built-in debug functionality

Goal: Standardizing interfaces between models and tools

Parameterizing a System



Generally, IPs come from many vendors

Parameter Examples

- system clock speed
- # processor cores
- memory size
- address, data widths
- disabled IP(s)
- address maps
- SW image filename
- IP granularity debug control*:
  - logging
  - tracing

* *runtime parameters provide initial CCI "control" capability*

Need uniform way to configure simulation without recompilation

CCI Environment

- CCI requires SystemC 2.3.1 and works better with 2.3.2
- In order to use CCI classes, a header must be included

```
#include <cci_configuration>
```

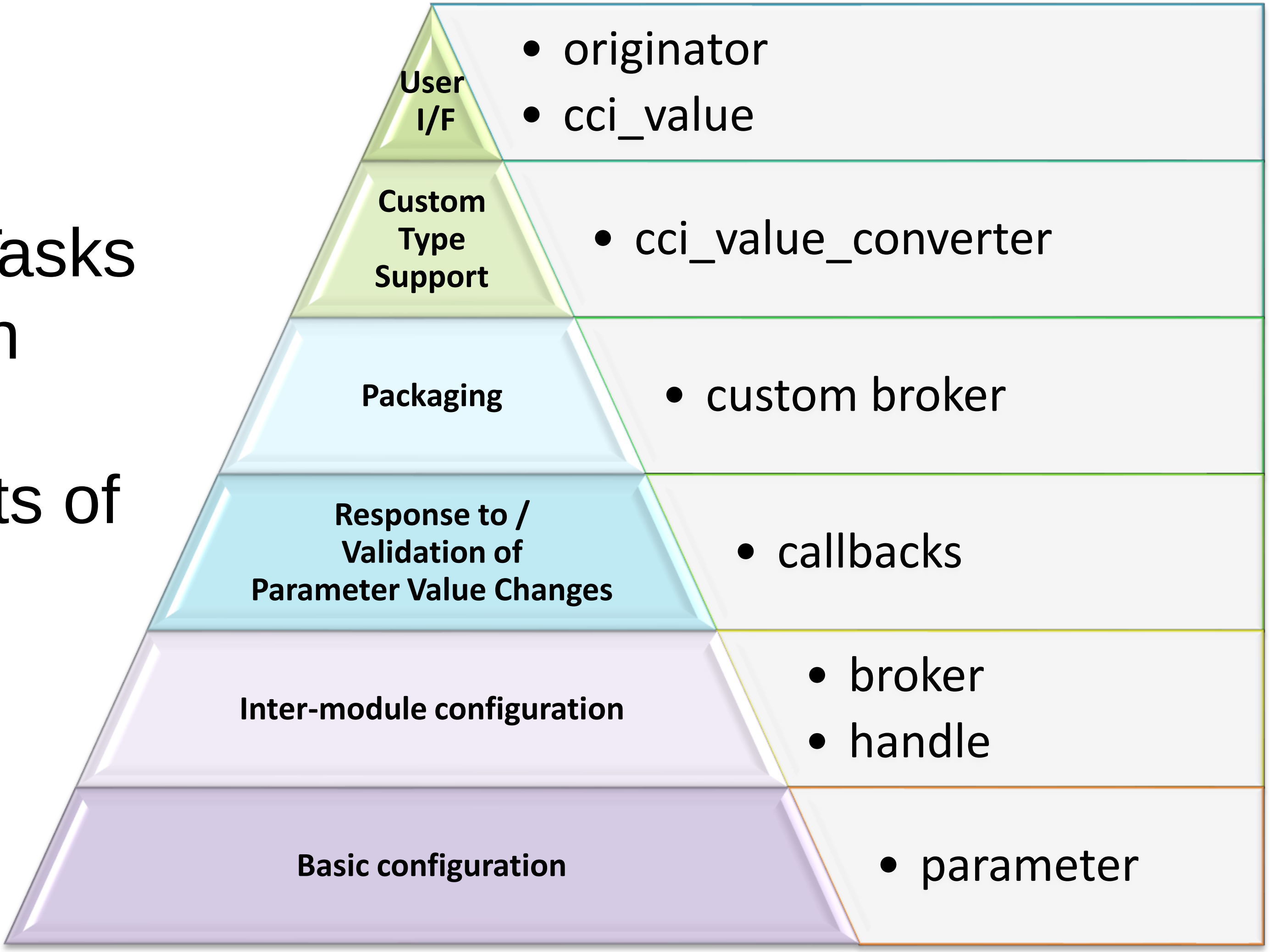
- As with SystemC, CCI code is defined in a specific namespace

```
namespace cci;
```

Configuration Overview

Configuration Tasks

- How common (relatively)
- Relevant parts of the standard



User I/F
- originator
- cci_value

Custom Type Support
- cci_value_converter

Packaging
- custom broker

Response to / Validation of Parameter Value Changes
- callbacks

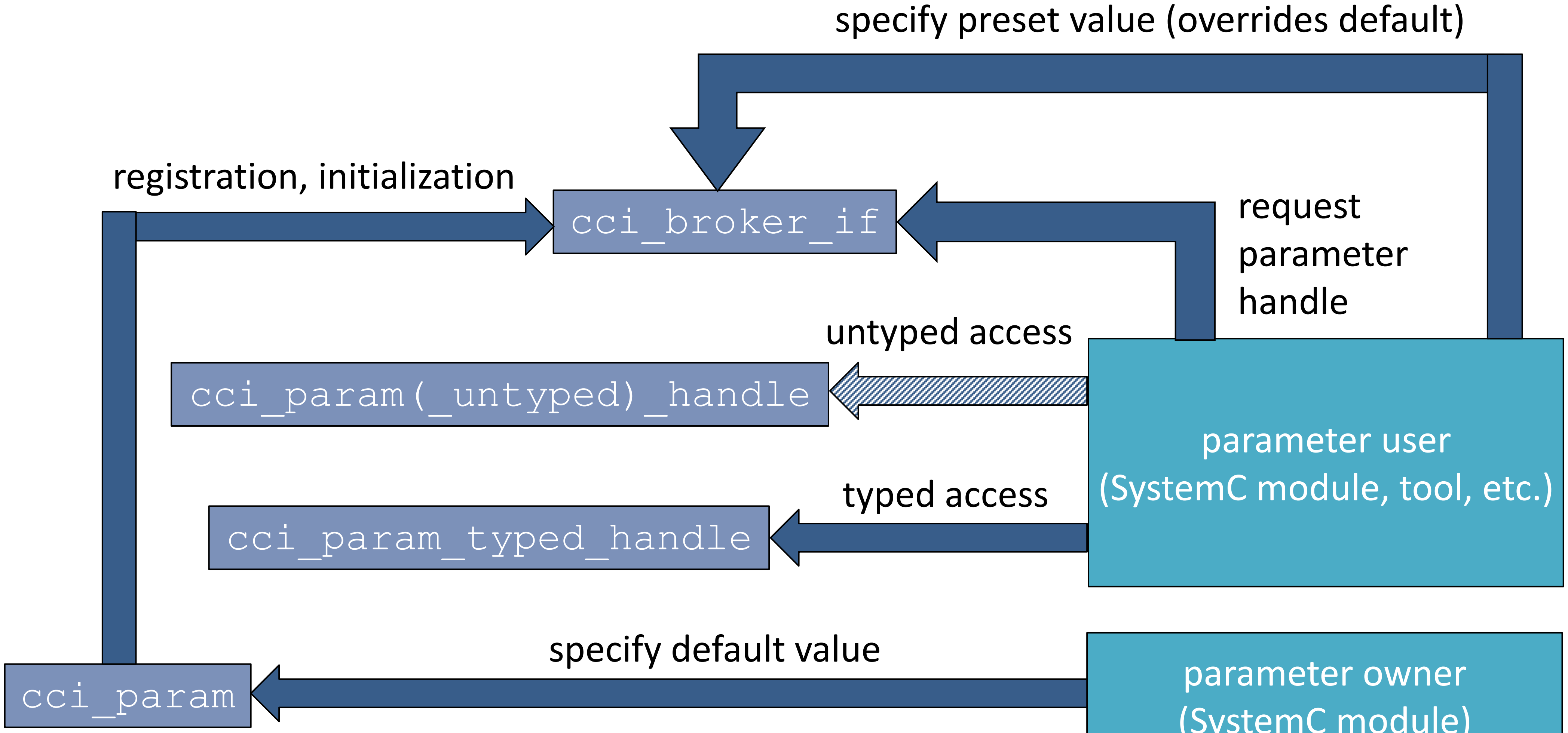
Inter-module configuration
- broker
- handle

Basic configuration
- parameter

Key Configuration Components

- Parameter
  - consists primarily of a name (string) and a value
  - is an instance of `cci_param<T>` (`T` is the value type)
  - registers with a broker at construction
  - provides 2 interfaces to set/get values
    - “untyped”: uses variant type; interoperable with JSON
    - “typed”: a templated interface using instantiated type T
- Broker
  - Manages access to parameters registered with it
    - Used by both models and infrastructure/tools
  - Two kinds of brokers:
    - There is one common global broker
    - Any number of custom brokers may also exist

Key Classes & Interactions



Default and Preset Values

- The broker allows a PRESET parameter value to be specified prior to the parameter's construction:

```
cci::cci_get_broker().set_preset_cci_value(
                      "param_name", cci::cci_value(10));
```

- When you create a parameter, you must specify a DEFAULT value:

```
cci::cci_param<int> my_param("param_name", 42);
```

- The parameter will use the PRESET value if it exists, otherwise it will use the DEFAULT value.

```
std::cout << my_param.get_value(); // Output = 10
```

Actual vs. Variant Value Types

There are two ways to access parameter values:

- **<u>Untyped</u>**: using a variant type, `cci_value`
  - Complex types emulated as collection of primitive types
    - Built-in support for basic and SystemC type conversions
    - Extensible to support user-defined types
  - Important for tool enabling, for example:
    - Applying preset values from an ASCII configuration file
    - Presenting/validating values of complex parameter types
- **<u>Typed</u>**: using the template instantiated value type
  - More direct (and efficient) when value type is known

Note: `cci_value` *may be promoted to core language as* `sc_variant`.

Originator

The origin of a parameter's current value is always known; the `cci_originator` class is used for this

- Within the SystemC module hierarchy, originator modules are determined automatically
  - e.g. "top.platformX.subsystemA.dma1"
- Identifying strings are provided outside the SystemC module hierarchy
  - e.g. "platformX_basic_configuration.cfg"
    Indicating the value came from a configuration file
  - e.g. "sim_user"
    Indicating the value was set interactively

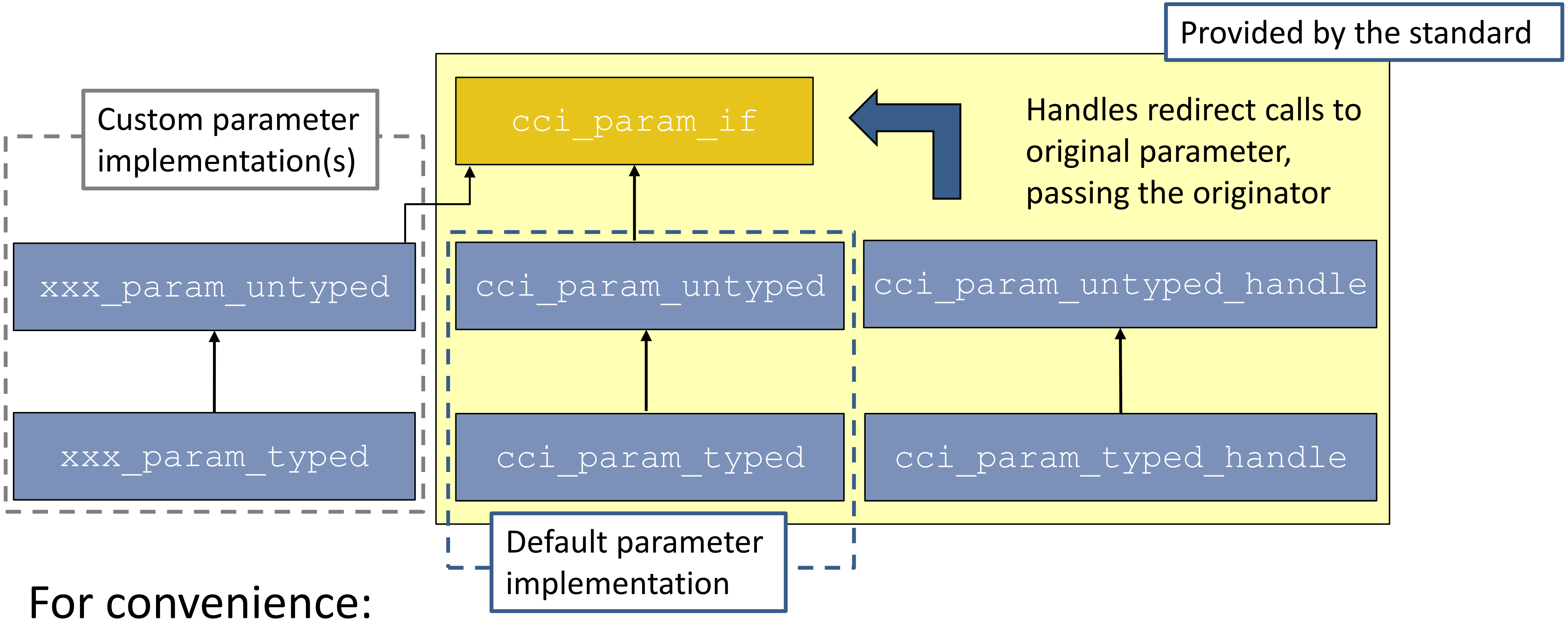
Originators are generally managed behind-the-scenes

Parameter Handles

- Returned by a broker for name-based parameter lookup
- Provides a parameter-like interface
- Informs parameter of originator when value is updated
- Available in both untyped and typed forms:

```
class cci_param_untyped_handle;
template<typename T> class cci_param_typed_handle;
```

Parameter and Handle



For convenience:

```
typedef cci_param_untyped_handle cci_param_handle
template<typename T> using cci_param = cci_param_typed<T>;
(pre-C++11: #define cci_param cci_param_typed)
```

Standard vs Implementation

- Standard:
  - Parameter and broker interface
  - Default implementation of `cci_param`
  - Originator and Handle
  - Callback Infrastructure
  - `cci_value`
- Vendor Implementation:
  - Specialized brokers
  - Support for configuration files (xml, conf…)

Note: the POC implementation provides example brokers in cci_utils/

A parameter owner module

```
SC_MODULE(simple_ip) {

private:
 cci::cci_param<int> int_param;

public:
 SC_CTOR(simple_ip) : int_param("int_param", 0)
 {
   int_param.set_description("...");
   SC_THREAD(do_proc);
 }
 void do_proc() {
   for(int i = 0; i < int_param; i++) {
     ...
   }
 }
};
```

Parameters are usually private members forcing brokered access

Default values must be supplied using the constructor argument

Owner reads parameter’s value

Accessing Parameter Values

- When value type is known, call parameter's `set_value(T val)` or `get_value()` function
  - Common C++ types
  - SystemC Data types
  - Extensible to user-defined types
- When parameter type is unknown or unsupported:
  - Use `cci_value` representation (variant type)
  - `set_cci_value()`
    - Takes variant typed value; fails if incompatible contents
  - `get_cci_value()`
    - Returns variant typed value

Broker Lookup of Params (1)

```
SC_MODULE(configurator)  {

 cci::cci_broker_handle m_brkr;


 SC_CTOR(configurator)
 : m_brkr(cci::cci_get_broker())
 {
   sc_assert(m_brkr.is_valid());

   SC_THREAD(do_proc);
 }
…
```

Declaration of broker handle variable

Get handle to broker associated with this module

Broker Lookup of Params (2)

```
void do_proc() {
 const std::string int_param_name = "top.sim_ip.int_param";

 cci::cci_param_handle int_param_handle =
     m_brkr.get_param_handle(int_param_name);

 if(int_param_handle.is_valid()) {

   cci::cci_value value = int_param_handle.get_cci_value();
   value = value.get_int() + 1;
   ...
   int_param_handle.set_cci_value(value);

   ...
  }
```

Get handle to named parameter from broker

Check handle validity / parameter exists

Get current parameter value

Set new parameter value

Parameter Mutability

- Parameters are mutable by default
- Mutability is set by template parameter

```
cci::cci_param<int, CCI_MUTABLE_PARAM> p1;
```

- Parameters may also be made immutable

```
cci::cci_param<int, CCI_IMMUTABLE_PARAM> p2;
```

- Locking can be used to make a mutable parameter temporarily immutable
  - E.g. to prohibit post-elaboration changes to parameters affecting design structure

Description and Metadata

- CCI parameters have a description intended to explain their purpose and usage to a simulation user.
- Supplied either as a constructor argument or with a setter.

```
my_param.set_description("Clock frequency");
```

- CCI parameters have array of meta information (cci_value_map)
- Can use this for any purpose

```
my_param.add_metadata("units", "V",
                      "Units of the value");
my_param.get_metadata(); // Return CCI values
```

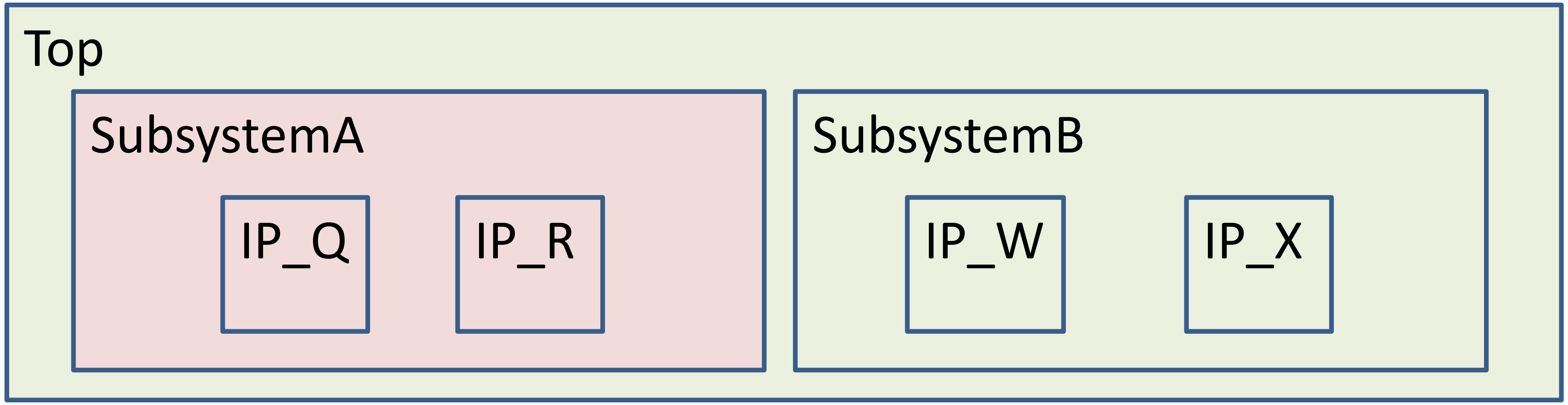
Brokers

- A broker may be registered with global scope or at a specific point in the module hierarchy
  - Where no broker is explicitly registered, the parent’s broker is inherited
- A broker must be in place prior to constructing parameters
  - Parameters are associated with the reigning broker
- Module level brokers are undiscoverable outside of their associated module hierarchy (“private” in nature)
  - Parameters are therefore inaccessible from outside that hierarchy
- Module level brokers facilitate encapsulation of IP configuration
  - E.g. a configurator that applies pre-compiled configuration

Broker Example

```
SC_CTOR(SubsystemA)
 {
   cci_register_broker(my_priv_broker);
 }
```

Only module itself can specify its broker



🟩 parameters managed by my_global_broker (registration not shown)

🟥 parameters managed by my_priv_broker

Callbacks

- Used to track parameter value changes: pre-read, post-read, pre-write and post-write
- Parameter creation and destruction callbacks are also available through the broker
- Callbacks contain a payload and can return a value
- Callback payloads can be typed or untyped
- Compatible with C++11 lambdas, function objects
- Internal callback mechanism provided by the standard

Note: The callback mechanism may be provided more widely across SystemC in the future.

Parameter Owner Callback

```
SC_MODULE(simple_ip) {
private:
 cci::cci_param<int> P1;
 cci::cci_callback_untyped_handle P1_cb;

public:
 SC_CTOR(simple_ip): P1("P1", 0) {
  P1_cb = P1.register_post_write_callback(&simple_ip::cb,this);
   ...
 }
 void cb(const cci::cci_param_write_event<int> & ev);
...
```

Callback handle

Post-write callback registered in constructor

Post-write callback function with the write event as parameter

Callback Events

- Callback `pre_write` and `post_write` event:
  - Contains an untyped parameter handle, the old value, new value and the originator
- Callback `pre_read` and `post_read` event:
  - Contains an untyped parameter handle, the value and the originator
- Callback `create_param` and `destroy_param` event:
  - Contains the untyped parameter handle
- Support for lambdas/function objects/`sc_bind` allows customization for different signatures and stateful callbacks

Custom types

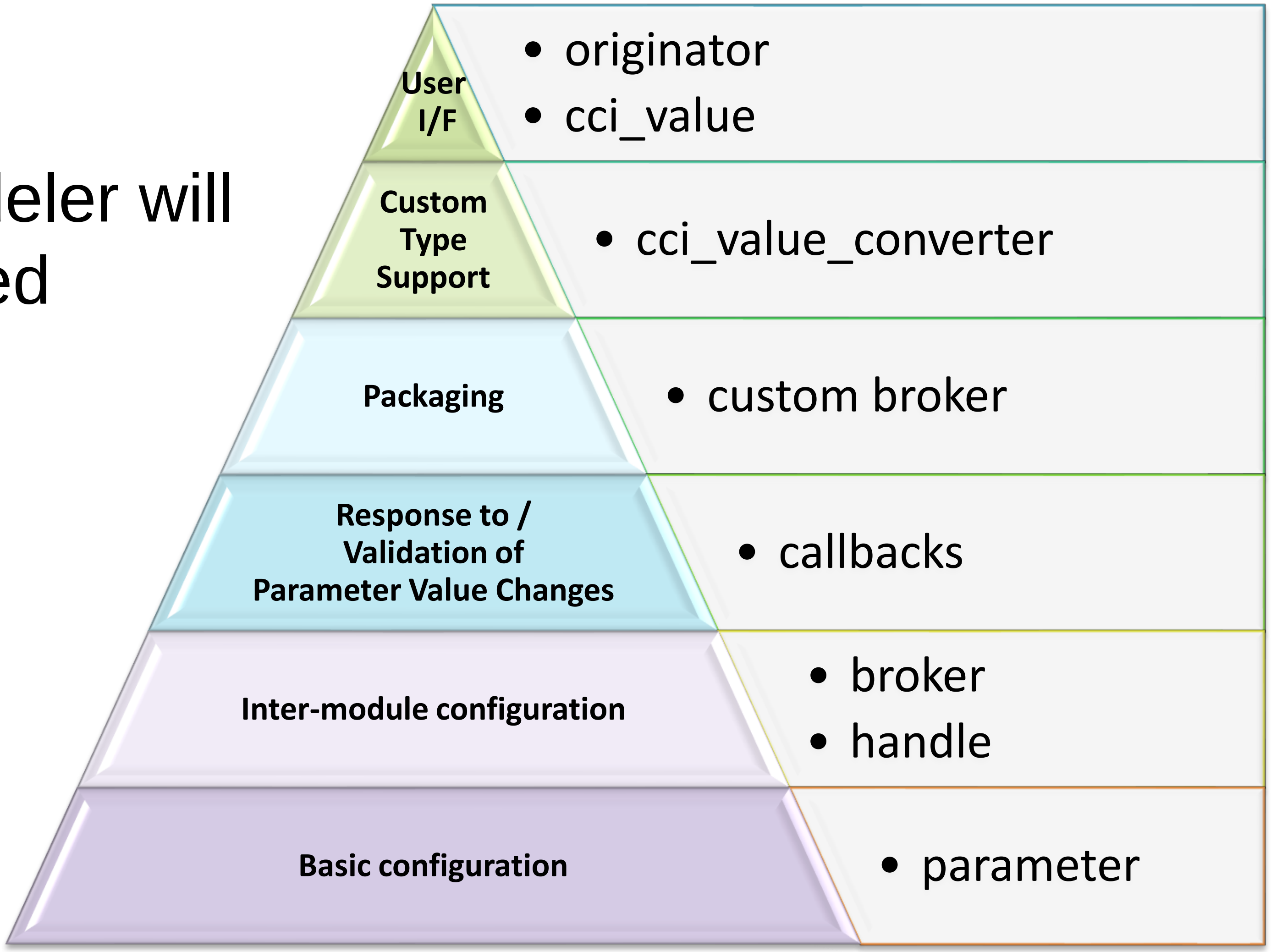
- Support for legacy parameter implementation is done via the `cci_param_if` interface
  - Explicit registration with the broker is required

- `cci_value` provides an extensible infrastructure to add packing/unpacking support for custom C++ datatypes.

- When a custom C++ data type is extended with `cci_value` support, it can be transparently used with `cci_param`

Summary Usage

The typical modeler will use only a limited subset of the standard on a routine basis.



User I/F	- originator - cci_value
Custom Type Support	- cci_value_converter
Packaging	- custom broker
Response to / Validation of Parameter Value Changes	- callbacks
Inter-module configuration	- broker - handle
Basic configuration	- parameter